Functional Programming in Java (Functional Interface)

Java8의 (순수) 함수형 프로그래밍

- 순수성 : 함수의 결과 값의 유일성
  - No side-effect: 한 수식은 문맥에 관계 없이 언제나 유일한 값을 가짐
  - 전역 변수 사용 금지: 함수의 계산에 영향을 주는 모든 정보는 인수로서 명시적으로 표현되어야 하며, 인수 이외의 정보가 함수의 계산에 영향을 주지 않음.
- Mutable / Immutable 자료 구조
  - 기존의 변수, 배열, 리스트 등의 자료구조는 모두 mutable함.
  - Final로 정의된 자료 구조는 immutable함.
  - 순수 함수형 프로그래밍은 immutable 자료 구조을 이용함.
  - Java에서는 final 변수와 stream을 적용함.
- 순수(pure)와 비순수(pure) 함수의 구분
  - 기존 비순수 형태의 함수 (메소드)는 그대로 사용하면서
  - 순수 함수형 프로그래밍은 Functional Interface 하에서 순수 함수를 정의함.
  - 결과적으로 타입으로 순수와 비순수를 구분함.

값의 유일성, Mutable / Immutable 자료 구조

- 할당문 (immutable)
  - 예) x = x + 2; (수학적으로 등식 관계가 성립하지 않으므로 의미 없는 식)
- 코드의 반복 수행을 위해서 loop (while, for 문 등) 을 사용하지 않음
  - 대신 재귀 함수 (recursive functions) 을 사용함
- 값의 유일성:
  - 한 수식의 값은 문맥에 상관없이 유일한 값을 갖으므로, 치환(substitution)에 의하 계산을 함.
  - 예를 들어, f(2) = 5 일 때, 임의 의 함수 g에 대해서 g(f(2)) = g(5) 가 성립함.
- Mutable / Immutable 자료 구조
  - Mutable : 변수, List 등.
  - Immutable : Python의 Tuple, Java의 stream.

- 함수 정의
  - 함수 이름, 인수의 순서 및 개수, 타입 표현

```
T funName(T1 a1, T2 a2, …, Tn an) {
    Function Body
    return val;
}
```

- 함수의 호출 (파라메터 패싱)

```
T x = funName(exp1, exp2, …., expn)
```

- 코드의 재사용
  - 한번 정의된 함수를 호출에 의해 재 사용
- 예

```
static boolean lessThan3 (int x) {
    return (x < 3);
}
```

[1,2,3,4] . map(lessThan3) (고차함수)
= [True, True, False, False]

- 람다식 (이름없는 함수)
  - 인수의 순서 및 개수, 타입 표현

```
Function<Integer, Boolean> lessThan3
   = x -> x < 3  ;
```

- 함수의 호출 (파라메터 패싱)

```
lessThan3.apply(4)
```

[1,2,3,4] . map(x -> x<3) = [True, True, False, False]

고차 함수의 개념 (함수가 First-class Citizen)

- 함수를 수식으로서 표현 (즉, 함수를 마치 산술식처럼 사용)
- 함수가 할당문의 오른쪽에 나타날 수 있음

```
Boolean x = x -> x < 3;
```

- 함수가 파라메터에 나타날 수 있음

```
static … f (Function<T,R> k) { … }
```

- 함수를 저장할 수 있음

```
List<Function<Integer,Boolean>> functions = [x -> x>2, x -> x==3, x ->
even(x) ]
```

- 결과 값이 함수가 될 수 있음

```
static boolean fun( ) { return x -> x <2; }
```

- 주의 : Java에서는 위의 표현이 어느 곳에서나 표현될 수는 없으며, 제한적인 형태로 표현될 수 있음.

자바에서 (순수) 함수형 프로그래밍 하기

- Java는 객체지향 프로그래밍
  - 기본적으로 mutable 자료 구조를 사용함
  - Instance 메소드 호출 형태 m.f(x)
    - 객체 m이 실제로는 f 메소드의 인수이지만, 이것이 괄호안에 표시되지 않음
    - 객체 내에 여러 field를 노출하여 인수로 표시할 수 없음

    즉, static 메소드가 아닌 경우 `f(m1, m2, …, mk, x)` 형태로 호출할 수 없음
- 비순수성 (mutability) 과 순수성(immutability)을 어떻게 표현?
- 고차 함수 및 람다 식을 얼마나 자유롭게 표현할 수 있을까?
- 제한적인 형태의 순수 함수형 프로그래밍 표현
  - Functional Interface
  - Collection의 forEach
  - Stream

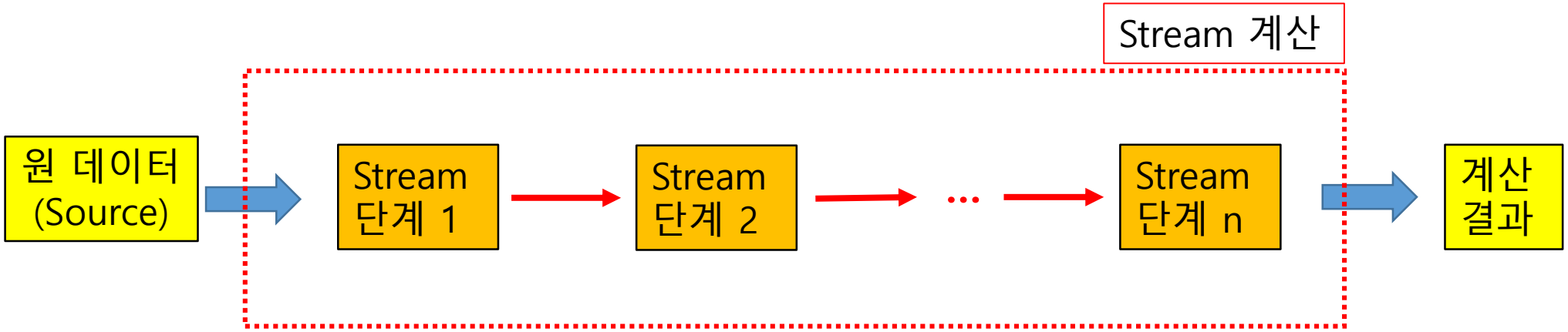
Functional Interface

- Annotation : @FunctionalInterface
- Java에서 순수 함수 식을 정의하기 위한 타입
- Functional Interface
  - 단 하나의 abstract method로만 구성됨
  - (추가적으로 default 메소드를 포함할 수도 있음)
  - Functional Interface는 람다 식과 method reference의 타입 으로 이용될 수 있음
- 람다 식 (함수)위한 타입 T (Functional Interface)

  T x = a -> …

람다 식(lambda expression) 사용 형식

1. Functional Interface
2. Stream 형태의 데이터 구조
   - Stream 형태로 전환
     - stream, of
   - Stream의 함수형 operators (pipeline 형태)
     - map, filter, sorted, 등의 함수 적용
   - Stream 종료 (다른 타입으로 전환)
     - reduce, collect, count, sum 등



람다 함수에 파라메터 전달

```
@FunctionalInterface
interface Square {
    int calculate(int x);
}

class FunInterface {
    public static void main(String args[]) {
        int a = 5;
        Square s = x -> x*x + a;
        int ans = s.calculate(a);
        System.out.println(ans);
        a = a + 1; // error
    }
}
```

1. Functional Interface 정의
   - 오직 하나의 추상 메소드
2. Interface 타입의 객체를 람다 식으로 정의 (함수)
   - 입출력 타입 일치
3. 함수 호출(파라메터 전송)
   - 인터페이스의 메소드 이용
4. 람다 식 내의 의 변수는 (effectively) final
   - Single assignment

java.util.function.* 패키지 이용

(https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)

- Function Interface

```
@FunctionalInterface
public interface Function<T,R>   {  R apply(T t); }
```

- 예

```
import java.util.function.Function;
Class Apply {
    public static void main(String args[]) {
        Function<Integer,Integer> square = x -> x*x;  // 람다 식
        int result = square.apply(5);                 // 파라메터 전송
        System.out.println(result);
}
```

이진함수에 대한 람다 식

```
@FunctionalInterface
interface M { int max(int x, int y); }          // binary

class Max {
    public static void main(String args[]) {
        M m = (a, b) -> a > b ? a : b;          //binary
        int result = m.max(10, 20);
        System.out.println(result);
    }
}
```

이진함수를 위한 BiFunction 인터페이스

- BiFunction Interface

```
@FunctionalInterface
public interface BiFunction<T,U,R>   {  R apply(T t, U u); }
```

예:

```
import java.util.function.BiFunction;
class Apply {
public static void main(String args[]) {
    BiFunction<Integer,Integer,Integer> max = (x,y) -> x>y ? x:y;
    int result = max.apply(3,5);
    System.out.println("Max is " + result);
}
```

No Input and No Return Value

```
@FunctionalInterface
interface N { void method1(); }

class Void {
    public static void main(String args[]) {
        N f = () -> System.out.println("No return value");
      f.method1();
    }
}
```

타입에 의한 인터페이스 구분

- 인터페이스에서 사용되는 generics는 void 타입을 표현하지 못함.
- 전송할 인수가 없거나, return 값이 없는 경우, generics를 이용하여 표현된 apply 메소드를 이용할 수 없음.

- 각 경우에 대한 독자적인 인터페이스와 추상 메소드를 패키지로 정의함.
  - Runnable 인터페이스 run 메소드 : 입력, 출력 모두 없음
  - Supplier 인터페이스 get 메소드 : 입력 없고, 출력은 있음
  - Consumer인터페이스 accept 메소드 : 입력은 있고, 출력 없음
  - Function 인터페이스 apply 메소드 : 입출력 모두 존재
  - Predicate 인터페이스 test 메소드 : 입출력이 모두 존재, 출력이 boolean으로 고정

인수의 개수와 타입에 따른 Interface

- 이진 함수에 따른 interface
  - `BiConsumer<T,U>,  BiPredicate<T,U>,  BiFunction<T,U,R>`
- 입력과 출력의 타입이 동일한 경우
  - `UnaryOperator<T>, BinaryOperator<T>`
- 기타
  - `DoubleToIntFunction, ToIntFunction <T>,  IntFunction<R>` 등

forEach : 함수를 파라메터로 전송

- Collection (Iterable 인터페이스) 의 forEach

  ```
  void forEach (Consumer<? super T> action)
  ```

  - forEach의 인수는 FI (Functional Interface) 의 instance (즉, 함수)
  - forEach의 인수로 람다 식과 method reference 를 사용할 수 있음
  - Inline implementation of functional interface

```
List<String> items = new ArrayList<>();
items.add("A");items.add("B");items.add("C"); items.add("D"); items.add("E");

for(String item : items)
    System.out.print(item + " " );

items.forEach(item -> System.out.print(item + " ")); // lambda

items.forEach (item -> {
    if("C".equals(item))
        System.out.print(item + " ");                // Output : C
});

items.forEach(System.out::print);                    // method reference

items.stream()                                       // Stream and filter, Output: B
        .filter(s->s.contains("B"))
        .forEach(System.out::print);
```

람다를 사용하는 재귀함수

```
import java.util.function.*;

public class FacRec {
    static final Function<Integer,Integer> factorial =
              x -> x == 0 ? 1 : x * FacRec.factorial.apply(x-1);

    final UnaryOperator<Integer> factorial2 = x -> x == 0   // instance
      ? 1 :  x * this.factorial2.apply(x-1);

    public static void main(String[] args) {
      System.out.println("factorial.apply(4) : " + FacRec.factorial.apply(4));
      FacRec a = new FacRec();  a.f();
    }

    void f() {  // instance functions are called within instance functions
      System.out.println("factorial2.apply(4) : " + factorial2.apply(4));       }
}
```

리스트 처리를 위한 재귀 함수

- Haskell 의 경우

```
sum xs = if x==[] then 0 else head(xs) + sum tail(xs)
```

주어진 리스트의 맨 앞 원소를 빼내고, 나머지 원소들을 가지고 새로운 리스트를 만들어서 재귀적으로 처리함 (리스트의 원소 수가 하나 감소됨)

- Java의 람다를 이용하는 재귀함수

```
static final Function<ArrayList<Integer>,Integer> sum = xs -> xs.isEmpty()
        ? 0
        : xs.get(0) + ListApply.sum.apply(new ArrayList<Integer>(xs.subList(1,xs.size())));

ArrayList<Integer> list1 = new ArrayList<Integer>() {{add(2);add(4);add(3);add(7);add(5);}};

Integer result = sum.apply(list2);
```